ru_vds
1 ноя 2018 в 16:00

Prettier, ESLint, Husky, Lint-Staged и EditorConfig: инструменты для написания аккуратного кода

16 мин 103K

Блог компании RUVDS.com, Разработка веб-сайтов*, JavaScript*, Совершенный код*

Перевод

Автор оригинала: Adeel Imran

Вы стремитесь к тому, чтобы писать аккуратный код, но не знаете с чего начать… Вы вчитываетесь в руководства по стилю, вроде этого от Airbnb, стараетесь следовать практическим рекомендациям ведущих специалистов… Вам приходится удалять неиспользуемый код? Приходится искать ненужные переменные? Вы пытаетесь выявлять неудачные паттерны, применённые в ваших программах? Например — хотите понять, читая хитросплетения кода некоей функции, возвратит ли она что-нибудь или нет. Звучит знакомо? Проблема заключается в том, что программисту очень тяжело и многое успевать, и многому учиться.

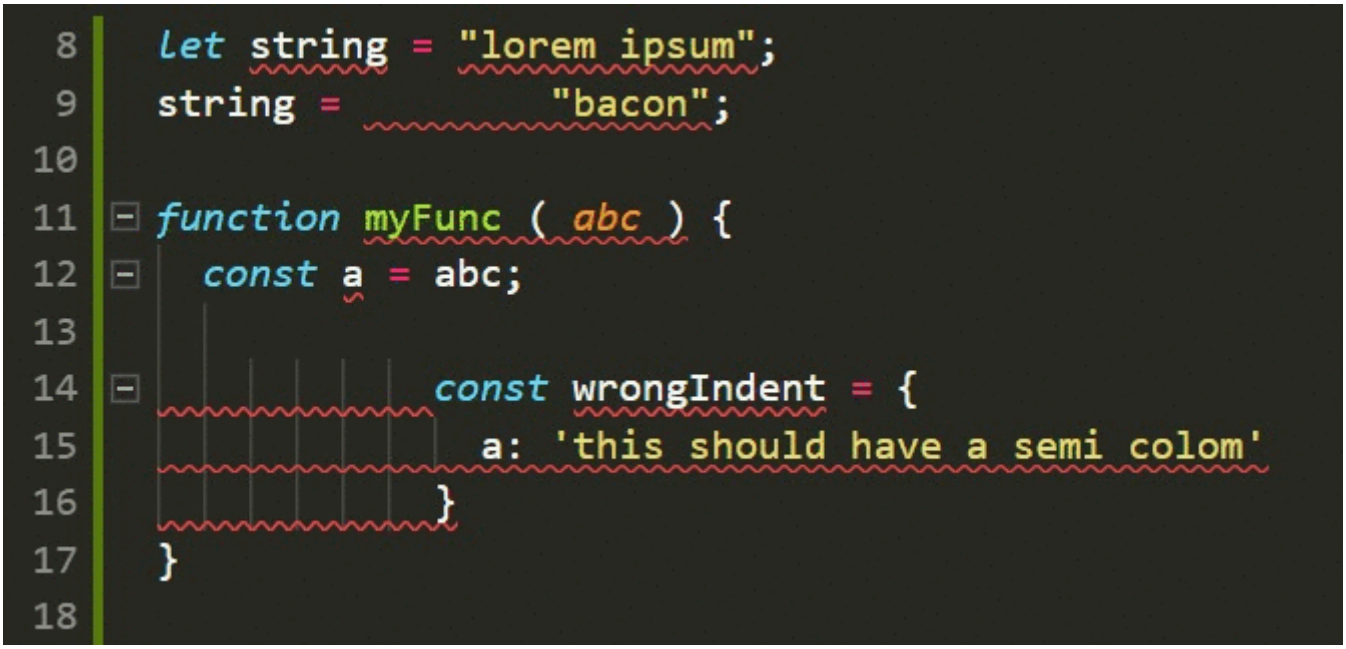
Может быть вы — тимлид, под началом которого трудится команда разработчиков разного уровня? В вашей команде есть новые люди? Беспокоит ли вас то, что код, который они напишут, не будет соответствовать вашим стандартам? Проходят ли ваши дни в проверках чужого кода, когда эти проверки, в основном, касаются соблюдения стандартов, а не программной логики?

Автор этого материала говорит, что он сталкивался со всем тем, чему посвящены только что заданные вопросы. То, с чем он столкнулся, утомляет и изматывает. Здесь он хочет рассказать об инструментах, правильное применение которых позволяет решить вышеописанные проблемы.

А именно, здесь пойдёт речь о таких средствах как Prettier, ESLint, Husky, Lint-Staged, EditorConfig, об автоматизации форматирования и линтинга кода. Этот материал ориентирован, в основном, на React-разработку, но рассмотренные здесь принципы можно применить в любом веб-проекте. Вот репозиторий, где, кроме прочего, собрано то, о чём тут пойдёт речь.

Prettier

Prettier — это средство для форматирования кода, которое нацелено на использование жёстко заданных правил по оформлению программ. Оно форматирует код автоматически. Вот как это выглядит.



Prettier форматирует код, следуя правилам

Сильные стороны Prettier

Вот какие возможности и особенности Prettier позволяют говорить о полезности этого инструмента:

- Приведение в порядок существующей кодовой базы. Подобное, с помощью Prettier, можно выполнить буквально одной командой. Ручная обработка больших объёмов кода займёт гораздо больше времени. Представьте себе, например, затраты труда, необходимые для того, чтобы вручную отформатировать 20000 строк кода.
- Prettier легко внедрить. Prettier использует «усреднённый», наименее спорный подход к стилю при форматировании кода. Так как проект это опенсорсный, многие внесли в него вклад, улучшая его и сглаживая острые углы.
- Prettier позволяет сосредоточиться на написании кода, а не на его форматировании. Многие просто не осознают того, как много времени и сил тратится на форматирование кода. Использование Prettier позволяет не думать о форматировании, а заниматься вместо этого программированием. В моём случае, например, эффективность работы, благодаря Prettier, выросла на 10%.
- Prettier помогает начинающим. Если вы — начинающий программист, работающий в одной команде с серьёзными профессионалами, и вы хотите достойно смотреться на их фоне, в этом вам поможет Prettier.

Настройка Prettier

Вот как использовать Prettier в новом проекте. Создайте папку `app`, и, перейдя в неё, выполните следующую команду в командной строке:

```
npm init -y
```

Благодаря этой команде `npm` инициализирует новый проект в папке `app`, создав в ней файл `package.json`.

Я, в этом материале, буду использовать `yarn`, но тут можно использовать и `npm`. Prettier можно подключить и к существующему проекту.

Установим пакет `prettier` в качестве зависимости разработки нашего проекта:

```
yarn add --dev prettier
```

Благодаря этой команде в `package.json` будет добавлена запись о зависимости разработки, которая выглядит так:

```
{
  "name": "react-boiler-plate",
  "version": "1.0.0",
  "description": "A react boiler plate",
  "main": "src/index.js",
  "author": "Adeel Imran",
  "license": "MIT",
  "scripts": {
    "prettier": "prettier --write src/**/*.js"
  },
  "devDependencies": {
    "prettier": "^1.14.3"
  }
}
```

О том, что означает строка `"prettier": "prettier—write src/**/*.js"`, мы поговорим чуть позже. А пока создадим в папке `app` папку `src`. В этой папке создадим файл `index.js`, хотя назвать его можно как угодно.

В этот файл внесём следующий код (именно в таком вот неприглядном виде):

```
let person =                    {
  name: "Yoda",
               designation: 'Jedi Master '
               };


             function trainJedi (jediWarrion) {
if (jediWarrion.name === 'Yoda') {
  console.log('No need! already trained');
}
console.log(`Training ${jediWarrion.name} complete`)
             }

trainJedi(person)
             trainJedi({ name: 'Adeel',
             designation: 'padawan'
  });
```

Итак, на данный момент у нас имеется файл `src/app/index.js`, в котором находится довольно-таки плохо оформленный код.

Как это исправить? Существует три подхода к работе с плохо отформатированным кодом:

1. Вручную отформатировать этот код.
2. Использовать автоматизированный инструмент.
3. Оставить всё как есть и работать дальше (прошу вас не выбирать этот подход).

Я собираюсь выбрать второй вариант. Сейчас в нашем проекте есть соответствующая зависимость, и, кроме того, в разделе `scripts` файла `package.json` есть запись о Prettier. Понятно, что мы воспользуемся для форматирования кода именно этим инструментом. Для того чтобы это сделать, создадим файл `prettier.config.js` в папке `app` и добавим туда правила для Prettier:

```
module.exports = {
  printWidth: 100,
  singleQuote: true,
  trailingComma: 'all',
```

```
  bracketSpacing: true,
  jsxBracketSameLine: false,
  tabWidth: 2,
  semi: true,
};
```

Разберём эти правила:

- `printWidth: 100` — длина строки не должна превышать 100 символов.
- `singleQuote: true` — все двойные кавычки будут преобразованы в одинарные. Подробности об этом можно почитать в руководстве по стилю от Airbnb. Мне очень нравится это руководство, я использую его для повышения качества моего кода.
- `trailingComma: 'all'` — обеспечивает наличие запятой после последнего свойства объекта. Вот хорошая статья на эту тему.
- `bracketSpacing: true` — отвечает за вставку пробелов между телом объекта и фигурными скобками в объектных литералах. Если это свойство установлено в `true`, то объекты, объявленные с использованием объектных литералов, будут выглядеть так: `{ foo: bar }`. Если установить его в `false`, то такие конструкции будут выглядеть так: `{foo: bar}`.
- `jsxBracketSameLine: false` — благодаря этому правилу символ `>` в многострочных JSX-элементах будет помещён в последней строке. Вот как выглядит код, если это правило установлено в `true`:

```
<button
  className="prettier-class"
  id="prettier-id"
  onClick={this.handleClick}>
  Click Here
</button>
```

Вот что произойдёт, если оно установлено в значение `false`:

```
<button
  className="prettier-class"
  id="prettier-id"
  onClick={this.handleClick}
```

```
>
  Click Here
</button>
```

- `tabWidth: 2` — задаёт количество пробелов на один уровень выравнивания.
- `semi: true` — если это правило установлено в `true`, то в конце выражений добавляется точка с запятой.

Здесь можно найти сведения по всем правилам Prettier.

Теперь, когда правила настроены, поговорим об этом скрипте:

```
"prettier": "prettier  — write src/**/*.js"
```

Благодаря этой конструкции Prettier запускается и находит все `.js`-файлы в папке `src`. Флаг `--write` указывает ему на то, чтобы он сохранял отформатированные файлы по мере их обработки и исправления найденных в них ошибок форматирования.

Запустим скрипт из командной строки:

```
yarn prettier
```

Вот что стало после этого с показанным выше плохо отформатированным кодом.

```
let person =                        {
  name: "Yoda",
                    designation: 'Jedi Master '
                    };


                function trainJedi (jediWarrion) {
if (jediWarrion.name === 'Yoda') {
  console.log('No need! already trained');
}
console.log(`Training ${jediWarrion.name} complete`)
                    }

trainJedi(person)
                trainJedi({ name: 'Adeel',
                designation: 'padawan'
  });
```

Результат форматирования кода с помощью Prettier

На этом будем считать, что с Prettier мы разобрались. Поговорим о линтерах.

ESLint

Линтинг — это вид статического анализа кода, который часто используют для нахождения проблемных паттернов проектирования или кода, который не следует определённым руководствам по стилю.

Существуют линтеры, предназначенные для большинства языков программирования, иногда компиляторы включают линтинг в процесс компиляции кода. Это определение линтинга взято со страницы информации об опенсорсном линтере для JavaScript ESLint, о котором мы и поговорим.

Зачем нужен линтер для JavaScript?

Так как JavaScript — это динамический язык программирования со слабой типизацией, код, написанный на нём, подвержен ошибкам, которые допускают разработчики. JavaScript — интерпретируемый язык, поэтому синтаксические и другие ошибки в коде обычно выявляются только после запуска этого кода.

Линтеры, наподобие ESLint, позволяют разработчикам находить проблемы в коде, не запуская его.

Почему ESLint — это особенный инструмент?

В заголовок этого раздела вынесен хороший вопрос. Дело тут в том, что ESLint поддерживает плагины. Так, правила проверки кода не должны представлять собой монолитный пакет. Всё, что нужно, можно подключать по мере необходимости. Каждое добавляемое в систему правило линтинга автономно, оно может быть, независимо от других, включено или выключено. Каждому правилу можно назначить уровень оповещения в соответствии с желанием разработчика — это может быть предупреждение (warning) или ошибка (error).

При использовании ESLint вы работаете с полностью настраиваемой системой, способной отразить ваше понимание того, как должен выглядеть правильный код, и зафиксировать то, какого свода правил вы придерживаетесь.

Среди существующих руководств по стилю JavaScript можно отметить следующие, весьма популярные:

- Google JavaScript Style Guide
- Airbnb JavaScript Style Guide

Я, как уже говорилось, использую руководство по стилям от Airbnb. Мне посоветовал этот документ мой руководитель в компании, в которой началась моя профессиональная карьера, и я считаю это руководство по стилям самым ценным своим активом.

Это руководство активно поддерживается — взгляните на его репозиторий на GitHub. Здесь я буду использовать набор правил, основанный именно на нём.

Сейчас давайте поработаем над файлом `package.json`, добавим в него некоторые зависимости:

```
{
  "name": "react-boiler-plate",
  "version": "1.0.0",
  "description": "A react boiler plate",
  "main": "src/index.js",
  "author": "Adeel Imran",
```

```
  "license": "MIT",
  "scripts": {
    "lint": "eslint --debug src/",
    "lint:write": "eslint --debug src/ --fix",
    "prettier": "prettier --write src/**/*.js"
  },
  "husky": {
    "hooks": {
      "pre-commit": "lint-staged"
    }
  },
  "lint-staged": {
    "*.(js|jsx)": ["npm run lint:write", "git add"]
  },
  "devDependencies": {
    "babel-eslint": "^8.2.3",
    "eslint": "^4.19.1",
    "eslint-config-airbnb": "^17.0.0",
    "eslint-config-jest-enzyme": "^6.0.2",
    "eslint-plugin-babel": "^5.1.0",
    "eslint-plugin-import": "^2.12.0",
    "eslint-plugin-jest": "^21.18.0",
    "eslint-plugin-jsx-a11y": "^6.0.3",
    "eslint-plugin-prettier": "^2.6.0",
    "eslint-plugin-react": "^7.9.1",
    "husky": "^1.1.2",
    "lint-staged": "^7.3.0",
    "prettier": "^1.14.3"
  }
}
```

Прежде чем рассказывать о том, как работать с этой конфигурацией, я хочу остановиться на зависимостях проекта, которые добавлены в `package.json`. Я полагаю, что, прежде чем использовать некие зависимости, стоит знать о том, какую роль они играют.

Поэтому обсудим роль представленных здесь пакетов:

- `babel-eslint` — позволяет использовать линтинг в применении ко всему тому, что даёт Babel. Этот плагин вам не нужен в том случае, если вы не используете Flow или экспериментальные возможности, которые пока не поддерживает ESLint.
- `eslint` — это основной инструмент, который используется для линтинга кода.

- `eslint-config-airbnb` — предоставляет правила Airbnb в виде конфигурации, которую можно модифицировать.
- `eslint-plugin-babel` — это плагин для ESLint, дополняющий плагин `babel-eslint`. В нём переделаны правила, которые, при применении `babel-eslint`, вызывают проблемы при обработке экспериментальных возможностей.
- `eslint-plugin-import` — этот пакет поддерживает линтинг свежих синтаксических конструкций `import/export` и позволяет предотвращать проблемы, связанные с неправильным написанием путей к файлам и имён импортируемых модулей.
- `eslint-plugin-jsx-a11y` — предоставляет правила, касающиеся доступности JSX-элементов для людей с ограниченными возможностями. Доступность веба — это очень важно.
- `eslint-plugin-prettier` — помогает совместной работе ESLint и Prettier. Выглядит это следующим образом: когда Prettier форматирует код, он делает это с учётом правил ESLint.
- `eslint-plugin-react` — содержит ESLint-правила, рассчитанные на React.

В этом материале мы не говорим о тестировании кода, но в представленном выше `package.json` есть зависимости, предназначенные для модульного тестирования с использованием Jest/Enzyme. Вот, если вы решите воспользоваться этими средствами для тестирования, описание соответствующих пакетов.

- `eslint-config-jest-enzyme` — данный пакет предназначен для тех случаев, когда пользуются `jest-environment-enzyme`, что приводит к тому, что переменные React и Enzyme оказываются глобальными. Благодаря ему ESLint не будет выдавать предупреждения о таких переменных.
- `eslint-plugin-jest` — ESlint-плагин для Jest.

В файле есть ещё пара пакетов, которые мы обсудим позже, обсуждая вопросы автоматизации. Это `husky` и `lint-staged`.

Теперь, когда мы, в общих чертах, обсудили наши инструменты, продолжим работу. Создадим файл `.eslintrc.js` в папке `app`:

```
module.exports = {
    env: {
```

```
        es6: true,
        browser: true,
        node: true,
    },
    extends: ['airbnb', 'plugin:jest/recommended', 'jest-enzyme'],
    plugins: [
        'babel',
        'import',
        'jsx-a11y',
        'react',
        'prettier',
    ],
    parser: 'babel-eslint',
    parserOptions: {
        ecmaVersion: 6,
        sourceType: 'module',
        ecmaFeatures: {
            jsx: true
        }
    },
    rules: {
        'linebreak-style': 'off', // Неправильно работает в Windows.

        'arrow-parens': 'off', // Несовместимо с prettier
        'object-curly-newline': 'off', // Несовместимо с prettier
        'no-mixed-operators': 'off', // Несовместимо с prettier
        'arrow-body-style': 'off', // Это - не наш стиль?
        'function-paren-newline': 'off', // Несовместимо с prettier
        'no-plusplus': 'off',
        'space-before-function-paren': 0, // Несовместимо с prettier

        'max-len': ['error', 100, 2, { ignoreUrls: true, }], // airbnb позволяет не
        'no-console': 'error', // airbnb использует предупреждение
        'no-alert': 'error', // airbnb использует предупреждение

        'no-param-reassign': 'off', // Это - не наш стиль?
        "radix": "off", // parseInt, parseFloat и radix выключены. Мне это не нрави

        'react/require-default-props': 'off', // airbnb использует уведомление об о
        'react/forbid-prop-types': 'off', // airbnb использует уведомление об ошибк
        'react/jsx-filename-extension': ['error', { extensions: ['.js'] }], // airb

        'prefer-destructuring': 'off',
```

```
        'react/no-find-dom-node': 'off', // Я этого не знаю
        'react/no-did-mount-set-state': 'off',
        'react/no-unused-prop-types': 'off', // Это всё ещё работает нестабильно
        'react/jsx-one-expression-per-line': 'off',

        "jsx-a11y/anchor-is-valid": ["error", { "components": ["Link"], "specialLin
        "jsx-a11y/label-has-for": [2, {
            "required": {
                "every": ["id"]
            }
        }], // для ошибки вложенных свойств htmlFor элементов label

        'prettier/prettier': ['error'],
    },
};
```

Теперь добавим в папку `app` файл `.eslintignore` :

```
/.git
/.vscode
node_modules
```

Поговорим теперь о том, как устроен файл `.eslintrc.js` , и о том, какой смысл несут представленные в нём конструкции.

Этот файл имеет следующую структуру:

```
module.exports = {
   env:{},
   extends: {},
   plugin: {},
   parser: {},
   parserOptions: {},
   rules: {},
};
```

Рассмотрим блоки этого файла, представленные объектами с соответствующими именами:

- `env` — позволяет задавать список сред, код для которых планируется проверять. В нашем случае тут имеются свойства `es6`, `browser` и `node`, установленные в `true`. Параметр `es6` включает возможности ES6 за исключением модулей (эта возможность автоматически устанавливает, в блоке `parserOptions`, параметр `ecmaVersion` в значение `6`). Параметр `browser` подключает глобальные переменные браузера, такие, как `Windows`. Параметр `node` добавляет глобальные переменные среды Node.js и области видимости, например — `global`. Подробности о средах можно почитать здесь.
- `extends` — представляет собой массив строк с конфигурациями, при этом каждая дополнительная конфигурация расширяет предыдущую. Здесь используются правила линтинга `airbnb`, которые расширены до `jest` и затем расширены до `jest-enzyme`.
- `plugins` — тут представлены правила линтинга, которые мы хотим использовать. У нас применяются правила `babel`, `import`, `jsx-a11y`, `react`, `prettier`, о которых мы уже говорили.
- `parser` — по умолчанию ESLint использует синтаксический анализатор Espree, но, так как мы работаем с Babel, нам надо пользоваться Babel-ESLint.
- `parserOptions` — так как мы изменили стандартный синтаксический анализатор на `babel-eslint`, нам необходимо задать и свойства в этом блоке. Свойство `ecmaVersion`, установленное в значение `6`, указывает ESLint на то, что проверяться будет ES6-код. Так как код мы пишем в `EcmaScript`-модулях, свойство `sourceType` установлено в значение `module`. И, наконец, так как мы используем React, что означает применение JSX, то в свойство `ecmaFeatures` записывается объект с ключом `jsx`, установленным в `true`.
- `rules` — эта часть файла `.eslintrc.js` нравится мне больше всего, так как она позволяет настраивать правила ESLint. Все правила, которые мы расширили или добавили с помощью плагинов, можно менять или переопределять, и делается это именно в блоке `rules`. В тексте файла имеются комментарии к правилам.

Теперь поговорим о файле `.eslintignore`. Этот файл принимает список путей, представляющий папки, содержимое которых не должно обрабатываться с помощью ESLint.

Здесь заданы три папки:

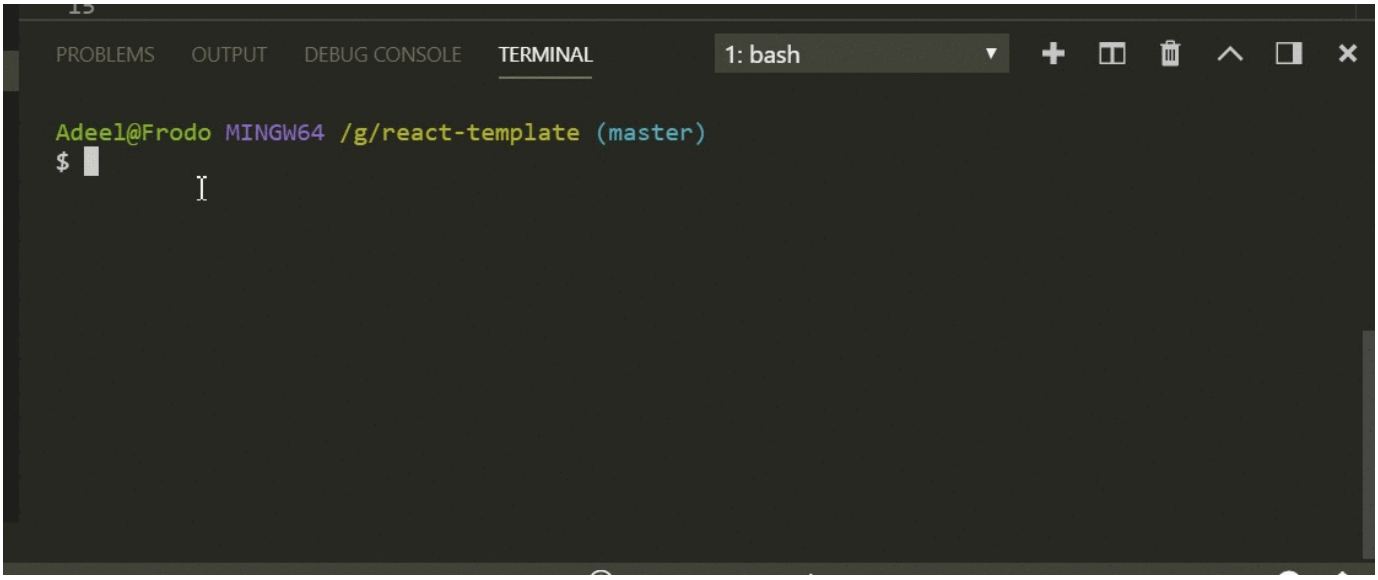
- `/.git` — мне не нужно, чтобы ESLint проверял файлы, относящиеся к Git.

- `/.vscode` — в проекте имеется эта папка из-за того, что я использую VS Code. Тут редактор хранит конфигурационные сведения, которые можно задавать для каждого проекта. Эти данные тоже не должны обрабатываться линтером.
- `node-modules` — файлы зависимостей также не нужно проверять линтером.

Сейчас рассмотрим пару новых скриптов, появившихся в `package.json`. Вот они:

```
"lint": "eslint --debug src/"
"lint:write": "eslint --debug src/ --fix"
```

Если выполнить первый из них, с помощью команды `yarn lint` или `npm run lint`, это приведёт к тому, что линтер просмотрит все файлы в директории `src` и выведет подробный отчёт по файлам, в которых он нашёл ошибки. Пользуясь этим отчётом можно эти ошибки исправить.



Запуск скрипта lint

Если выполнить второй скрипт (`yarn lint:write`), то ESLint выполнит такую же проверку, которая была выполнена раньше. Единственное различие заключается в том, что в таком режиме система попытается исправить обнаруженные ошибки, постарается привести код в как можно более пристойный вид.

Расширение ESLint для VS Code

У нас уже есть настроенные Prettier и ESLint, но, чтобы пользоваться возможностями этих инструментов, нам приходится запускать скрипты. Это не очень-то удобно, поэтому

попробуем это исправить. А именно, мы хотим добиться того, чтобы форматирование и линтинг кода выполнялись бы по команде сохранения файла в редакторе. Кроме того, выполнять линтинг и форматирование кода мы хотим перед выполнением коммитов.

Мы, в качестве примера, используем редактор VS Code. Нам понадобится расширение ESLint для VS Code. Для того чтобы установить его, можно открыть панель расширений VS Code (`ctrl+shift+x`). Тут, в поле поиска, надо ввести `eslint` . Появится список расширений. Нас интересует то, в сведениях о разработчике которого указан Dirk Baeumer. После установки этого расширения перезагрузим редактор.

Теперь, в корневой папке проекта (`app`), создайте папку `.vscode` (обратите внимание на точку в начале имени — это важно). В этой папке создайте файл `settings.json` следующего содержания:

```
{
  "editor.formatOnSave": false,
  "eslint.autoFixOnSave": true,
}
```

Рассмотрим его содержимое.

- Свойство `editor.formatOnSave` , установленное в значение `false` , указывает на то, что нам не нужно, чтобы стандартная конфигурация применялась бы к форматированию файла, так как это может вызвать конфликт с ESLint и Prettier.
- Свойство `eslint.autoFixOnSave` установлено в `true` , так как нужно, чтобы установленный плагин срабатывал бы каждый раз, когда сохраняют файл. Так как ESLint и Prettier в проекте работают совместно, сохранение файла приводит и к форматированию, и к линтингу кода.

Важно отметить, что теперь, когда запускается скрипт `lint:write` , он выполнит и линтинг и форматирование кода.

Представьте свои ощущения, если бы к вам попал код проекта размером в 20000 строк, который вам надо было бы проверить и улучшить. А теперь представьте себе, что вам пришлось бы это делать вручную. Такая работа заняла бы, наверное, месяц. А с помощь вышеописанных средств автоматизации всё это делается секунд за тридцать.

Теперь, после настройки всего необходимого, каждый раз, когда вы сохраняете файл с

кодом, редактор сам позаботится о проверке и форматировании текста программы. Однако тут мы говорим о редакторе VS Code. Вполне возможно, что кто-то в вашей команде предпочитает какой-нибудь другой редактор. Ничего плохого в этом нет, но, чтобы всем удобно было работать, нам придётся позаниматься ещё кое-что автоматизировать.

Husky

Пакет Husky позволяет задействовать хуки Git. Это означает, что у вас появляется возможность выполнять некие действия перед выполнением коммита или перед отправкой кода репозиторий.

Для того чтобы воспользоваться возможностями Husky, сначала установим этот пакет:

```
yarn add --dev husky
```

После этого добавим в `package.json` следующее:

```
"husky": {
   "hooks": {
     "pre-commit": "YOUR_COMMAND_HERE",
     "pre-push": "YOUR_COMMAND_HERE"
   }
},
```

Это приведёт к тому, что перед выполнением команды `commit` или `push` будет вызван некий скрипт, который, например, выполняет тестирование кода или его форматирование.

Подробности о Husky можно почитать здесь.

Lint-staged

Пакет Lint-staged позволяет проверять с помощью линтера индексированные файлы, что помогает предотвратить отправку в репозиторий кода с ошибками.

Линтинг имеет смысл проводить до коммита кода. Благодаря этому можно сделать так, чтобы ошибки не проникали в репозиторий, и обеспечить единую стилизацию кода, попадающего туда. Однако выполнение линтинга для проверки всего проекта может оказаться слишком длительной задачей, а результаты такой проверки могут оказаться бессмысленными. В конечном счёте, линтингу может понадобиться подвергнуть файлы,

которые планируется закоммитить.

Lint-staged позволяет выполнять набор произвольных задач над индексированными файлами, отфильтрованными по шаблону поиска. Подробности об этом можно почитать здесь.

Установим пакет Lint-staged:

```
yarn add --dev lint-staged
```

Затем, в файл `package.json`, добавим следующее:

```
"lint-staged": {
   "*.(js|jsx)": ["npm run lint:write", "git add"]
},
```

Благодаря этой конструкции сначала будет выполняться команда `lint:write`, производящая проверку содержимого файла и исправление ошибок, после чего файлы будут добавляться в индекс командой `git add`. Сейчас эта команда нацелена на `.js` и `.jsx`-файлы, но то же самое можно делать и с файлами других типов.

Совместное использование Husky и Lint-staged

Рассмотрим схему действий, которая позволяет организовать следующий рабочий процесс. Каждый раз, когда вы коммитите файлы с кодом, перед выполнением этой операции, система запускает скрипт `lint-staged`, который, в свою очередь, запускает скрипт `lint:write`, выполняющий линтинг и форматирование кода. После этого файлы добавляются в индекс, а затем коммитятся. Мне кажется, что это очень удобно. На самом деле, в ранее представленном коде файла `package.json` это уже реализовано, просто раньше мы об этом не говорили.

Приведём снова, для удобства, содержимое нашего `package.json`:

```
{
  "name": "react-boiler-plate",
  "version": "1.0.0",
  "description": "A react boiler plate",
  "main": "src/index.js",
```

```
  "author": "Adeel Imran",
  "license": "MIT",
  "scripts": {
    "lint": "eslint --debug src/",
    "lint:write": "eslint --debug src/ --fix",
    "prettier": "prettier --write src/**/*.js"
  },
  "husky": {
    "hooks": {
      "pre-commit": "lint-staged"
    }
  },
  "lint-staged": {
    "*.(js|jsx)": ["npm run lint:write", "git add"]
  },
  "devDependencies": {
    "babel-eslint": "^8.2.3",
    "eslint": "^4.19.1",
    "eslint-config-airbnb": "^17.0.0",
    "eslint-config-jest-enzyme": "^6.0.2",
    "eslint-plugin-babel": "^5.1.0",
    "eslint-plugin-import": "^2.12.0",
    "eslint-plugin-jest": "^21.18.0",
    "eslint-plugin-jsx-a11y": "^6.0.3",
    "eslint-plugin-prettier": "^2.6.0",
    "eslint-plugin-react": "^7.9.1",
    "husky": "^1.1.2",
    "lint-staged": "^7.3.0",
    "prettier": "^1.14.3"
  }
}
```

Теперь, зная о Husky и Lint-staged, вы можете оценить их влияние на работу с Git. А именно, предположим, что были выполнены следующие команды:

```
$ git add .
$ git commit -m "some descriptive message here"
```

Понятно, что перед коммитом код будет проверен на соответствие правилам, заданным в `.eslintrc.js`, и, при необходимости, исправлен. Благодаря этому ошибки никогда не проберутся в репозиторий рабочего проекта.

Теперь вы знаете о том, как интегрировать Prettier, ESLint, Husky и Lint-staged в свой проект.

Напомню, что выше мы говорили о том, что далеко не все члены вашей команды пользуются любимым мной VS Code. Для того чтобы всем им было удобно работать, нам понадобится разобраться с файлом `.editorconfig`.

Файл .editorconfig

Разные члены вашей команды могут использовать разные редакторы. Принуждать их использовать какой-то один редактор ни к чему. Однако для того, чтобы все пользовались едиными настройками, касающимися, например, отступов или символов перевода строки, мы и применяем файл `.editorconfig`. Он помогает поддерживать единый набор правил в неоднородных командах.

На сайте проекта можно найти список редакторов, которые поддерживают этот файл. В него, в частности, входят WebStorm, AppCode, Atom, Eclipse, Emacs, BBEdit и другие.

Создадим в папке `app` нашего проекта файл `.editorconfig` и добавим в него следующий код:

```
# EditorConfig - это замечательно: http://EditorConfig.org

# Файл EditorConfig верхнего уровня
root = true

[*.md]
trim_trailing_whitespace = false

[*.js]
trim_trailing_whitespace = true

# Переводы строк в стиле Unix с пустой строкой в конце файла
[*]
indent_style = space
indent_size = 2
end_of_line = lf
charset = utf-8
insert_final_newline = true
max_line_length = 100
```

Поясним настройки, использованные в этом файле:

- `trim_trailing_whitespace = false` — удаление пробелов в конце строк в `.md` -файлах не производится. Аналогичный параметр для `.js` -файлов установлен в `false`.
- `indent_style = space` — отступы оформляются пробелами а не знаками табуляции.
- `indent_size = 2` — размер отступа равен двум пробелам.
- `end_of_line = lf` — перевод строки оформляется символом `lf`. Это позволит всем, независимо от применяемых ими операционных систем, пользоваться одним и тем же символом перевода строки. Подробности об этом смотрите здесь.
- `insert_final_newline = true` — в конце файла должна быть пустая строка.
- `max_line_length = 100` — максимальная длина строки установлена в 100 символов.

Итоги

Полагаем, прочитав этот материал, вы вполне готовы к тому, чтобы создать удобную среду разработки в команде любого масштаба. Инструменты, представленные здесь, помогут вам поддерживать порядок в коде проектов и автоматизировать выполнение рутинных задач.

Уважаемые читатели! Какими инструментами вы пользуетесь для проверки и форматирования кода? Как автоматизируете эти процессы?

Теги: JavaScript, разработка

Хабы: Блог компании RUVDS.com, Разработка веб-сайтов, JavaScript, Совершенный код

325 Карма 546 Рейтинг

@ru_vds

Пользователь

Комментарии 15

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ

Кейс

+38 12 4

nkalacheva
16 часов назад

6 простых принципов написания приложения на Vue, которое легко поддерживать (часть 1)

8 мин 2.6K

Туториал

+37 79 24

maisvendoo
18 часов назад

Правда о железнодорожных тормозах: часть 5 — тормоза локомотивов

Средний 14 мин 4.3K

+30 16 2

Показать еще

Настройка языка

Техническая поддержка

Вернуться на старую версию

ИНФОРМАЦИЯ

Сайт	ruvds.com
Дата регистрации	18 марта 2016
Дата основания	27 июля 2015

Численность	11–30 человек
Местоположение	Россия
Представитель	ruvds

ССЫЛКИ

ПРИЛОЖЕНИЯ

ВИДЖЕТ

БЛОГ НА ХАБРЕ

18 часов назад

Учимся тестированию с помощью Bootstrap

1.4K 5

22 часа назад

Проверка XML. Schematron

620 1

13 мар в 21:00

О трудном и утомительном пути от идеи до веб-сайта

1.9K 2

13 мар в 16:00

Джон Кармак взялся за сильный ИИ — и у него особый подход. Список фундаментальной литературы для начала

17K 61

13 мар в 13:00

5 полезных библиотек Python (с примерами)

 8.4K 8